CEIG'08, Barcelona, Sept. 3-5 (2008)
L. Matey and J. C. Torres (Editors)

Voxelización de sólidos mediante instanciación de geometría

J.M. Noguera, A.J. Rueda, C.J. Ogáyar y R.J. Segura

Depto. de Informática, Escuela Politécnica Superior, Universidad de Jaén
Campus Las Lagunillas, Edificio A3, 23071 Jaén, Spain
{jnoguera, ajrueda, cogayar, rsegura}@ujaen.es

Resumen

En este artículo analizamos un algoritmo para calcular la voxelización de un sólido que es simple, eficiente, robusto y que aprovecha gran parte del potencial del hardware gráfico 3D. La nueva generación de hardware gráfico ha aportado innovadoras características tales como la instanciación de geometría y el procesador de geometría. Describimos cómo estas nuevas características pueden emplearse para implementar de manera sencilla y eficiente un algoritmo de voxelización. Se consigue así reducir el tiempo necesario para subir el modelo a la GPU. Se presentan dos implementaciones del algoritmo. La primera hace uso de la instanciación de geometría, y la segunda emplea conjuntamente la instanciación y el procesador de geometría.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Graphics processors I.3.5 [Computer Graphics]: Curve, surface, solid, and object representation

1. Introducción

La representación basada el vóxeles tiene una gran importancia en aplicaciones tales como visualizaciones médicas o para visualizar objetos que difícilmente se pueden representar a partir de sus fronteras, tales como humo o fuego. Además los vóxeles son entidades tridimensionales que pueden almacenar información volumétrica, al contrario que las representaciones basadas en fronteras (tales como mallas) que tan solo describen la superficie de los objetos. Las representaciones basadas en vóxeles ofrecen información de cómo se dispone la materia en el espacio.

La voxelización de un sólido consiste en convertir un objeto desde su representación geométrica continua a un conjunto de vóxeles que lo aproximen [KCY93]. En [ORSF07] se describe un método de voxelización basado en GPU capaz de trabajar con una amplia variedad de objetos poliédricos, con o sin agujeros, con autointersecciones y que es capaz de calcular la voxelización del interior del objeto. En este artículo describirmos cómo se puede extender este método para explotar las nuevas características presentadas en la serie 8 de GPUs NVIDIA GeForce: la instanciación de geometría y el procesador de geometría.

El resto del artículo se estructura de la siguiente manera. La sección 2 resume la literatura existente sobre técnicas de voxelización aceleradas por hardware. La sección 3 describe el trasfondo teórico y el algoritmo básico de voxelización empleado. En la sección 4 se explica el concepto de instanciación de geometría y procesador de geometría. La sección 5 se centra en describir las diferentes implementaciones aceleradas por hardware del algoritmo. Los resultados experimentales se muestran en la sección 6. Por último, en la sección 7 se exponen las conclusiones de nuestro trabajo.

2. Trabajo previo

En los últimos años se ha comenzado a emplear el hardware gráfico para calcular rápidamente la voxelización de sólidos. Tanto Fang y Chen [FC00] como Karabassi et al. [KPT99, PKT04] ofrecen métodos acelerados por hardware que calculan la voxelización en el interior del sólido. Por desgracia, no funcionan

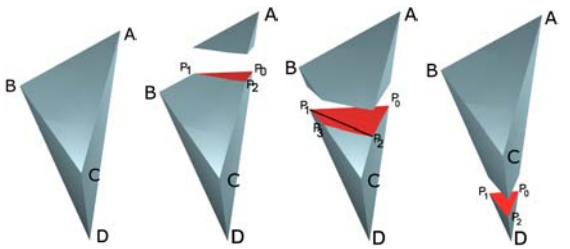

Figura 1: *División de un tetraedro en lonchas.*

en cualquier tipo de sólidos, por lo que no pueden considerarse métodos de voxelización generales.

Otros autores como Li et al. [LFWK05] o Eisemann [ED06] ofrecen algoritmos de voxelización por hardware muy rápidos, pero que en cambio tan solo voxelizan la frontera del objeto, y no su interior.

Rueda et al. [ORSF07] propuso un algoritmo de voxelización robusto que puede ser implementado eficientemente en la GPU.

3. Algoritmo de voxelización

La base teórica del algoritmo de voxelización reside en el test de inclusión de Feito et al. [FT97].

Lema 3.1 Dado un punto origen arbitrario O y un poliedro G definido por sus caras triangulares $f_0, f_1, ..., f_n$, y sea $S = T_1, T_2, ..., T_n$ el recubrimiento de G mediante símplices 3D (tetraedros) T_i definidos por O y la cara triangular f_i. Entonces un punto P estará dentro de G si está incluído en un número par de tetraedros de S. Este lema se demuestra en [ORSF07].

El algoritmo de voxelización de poliedros definidos por caras triangulares consiste en establecer un punto como el origen O (generalmente el centroide), construir el conjunto de tetraedros S y rasterizarlos en un *buffer de presencia*. El estado final de este *buffer* representa la voxelización del poliedro.

Un *buffer* de presencia es un vector 3D de *valores de presencia* con la misma dimensión que el espacio de vóxeles. Cada voxel tiene asociado un valor de presencia, que se representa mediante un bit. Un valor 1 significa que el voxel pertenece al sólido, mientras que 0 significa lo contrario. La rasterización de los tetraedros se realiza invirtiendo todos los valores de presencia recubiertos por el tetraedro. El lema 3.1 garantiza que el contenido final del buffer es la voxelización del objeto.

En [ORSF07] se propuso un algoritmo que consiste en procesar sucesivas lonchas de un tetraedro. Sea

Punto	$A_y > y_s \geq B_y$	$B_y > y_s \geq C_y$	$C_y > y_s \geq D_y$
P_0	$\overline{AD}$	$\overline{AD}$	$\overline{AD}$
P_1	$\overline{AB}$	$\overline{BD}$	$\overline{BD}$
P_2	$\overline{AC}$	$\overline{AC}$	$\overline{CD}$
P_3	-	$\overline{BC}$	-

Tabla 1: *Aristas empleadas para la interpolación de puntos dependiendo de la posición relativa del plano de corte.*

$\triangle ABCD$ un tedraedro arbitrario. El método consiste en cuatro pasos:

1. Escoger una dirección para los cortes. Asumiremos que el corte en lonchas se realiza desplazando el plano a lo largo del eje y. Ordenar los vértices del tetraedro por sus coordenadas y. Sea A el vértice con mayor coordenada y, B el siguiente y así con C y D (ver figura 1a). El proceso comienza con $y_s = A_y$ y termina cuando $y_s = D_y$.
2. Procesar las intersecciones del tetraedro con el plano de corte actual. Notaremos estos puntos como P_0, P_1, P_2, P_3, como se muestra en la figura 1b. Estas intersecciones se pueden procesar por una simple interpolación lineal o aplicando un cálculo incremental. La tabla 1 muestra las aristas que se deben emplear para procesar estos puntos, dependiendo del valor de y_s. Nótese que el punto P_3 tan solo aparece en el intervalo $B_y > y_s \geq C_y$, tal y como se ve en la figura 1c.
3. Voxelizar la loncha y_s del tetraedro. Esto se puede realizar mediante un escaneado del triángulo $\triangle P_0P_1P_2$ mostrado en la figura 1b. En el intervalo $B_y > y_s \geq C_y$, un segundo triángulo $\triangle P_3P_2P_1$ se debe de procesar (ver figura 1c). Durante esta operación, los valores de presencia de todos los vóxeles x, y_s, z cubiertos por los triángulos se deben intercambiar.
4. Decrementar y_s y repetir los pasos 2 y 3 hasta que $y_s = D_y$.

4. Procesador de geometría e instanciación

El procesador de geometría [BL06] es una nueva característica de la serie 8 de NVIDIA GeForce, que permite la creación dinámica de geometría en la GPU. Hasta ahora, el hardware gráfico solo tenía la habilidad de manipular datos ya existentes en la GPU. Pero con la llegada del procesador de geometría, la GPU es capaz de crear y destruir vértices en el procesador de vértices. Esto permite la creación de nuevas primitivas gráficas, tales como pixels, líneas y triángulos a partir de aquellas primitivas que fueron enviadas al principio del cauce gráfico.

Los programas del procesador de geometría se ejecutan después del procesado de vértices y toman como entrada una primitiva completa. Por ejemplo, cuando trabajamos con triángulos, los tres vértices son enviados como entrada al procesador de geometría. Éste puede emitir cero o más primitivas, que son rasterizadas hasta que finalmente sus fragmentos son enviados al procesador de fragmentos.

Usos típicos del procesador de geometría incluyen la teselación de geometría, cálculo de sombras volumétricas generadas en la GPU, etc.

Por otro lado, la instanciación de geometría [Car05] permite el procesado de múltiples instancias de un objeto con una única llamada a una función de dibujado. Es decir, se envía una única malla a la GPU, y ésta se encarga de replicarla tantas veces como se le solicite. Cada uno de los vértices así generados provoca una ejecución del procesador de vértices. A tal fin, OpenGL proporciona una extensión para trabajar con esta característica (*GL_EXT_draw_ instanced*) [Gol06]. Mediante esta extensión podemos emplear una nueva función, llamada *DrawArraysInstancedEXT*. Esta función se comporta de forma análoga a *DrawArrays* con la diferencia de que admite un nuevo parámetro que indica el número de instancias a dibujar.

Toda la información de los vértices está duplicada para cada malla instanciada, por lo que es necesario procesar individualmente cada una de dichas mallas para producir el efecto de repetición requerido. Para ello, a cada instancia se le proporciona una única variable identificadora de instancia (llamada *gl_InstanceID* en GLSL) que puede ser empleada para procesar de forma distinta cada instancia, generalmente aplicando una transformación.

Esta técnica se emplea principalmente para objetos tales como árboles, hierba o edificios que pueden ser representados mediante la misma geometría sin parecer toscamente repetidos.

5. Implementaciones del algoritmo

5.1. Implementación en GPU previa

La solución en GPU propuesta en [ORSF07] emplea un programa de vértices que calcula las posiciones de los puntos P_0, P_1, P_2, P_3 para un tetraedro y una loncha dados. Para ello se emplea interpolación lineal en función de la arista que corresponda, tal y como se muestra en la tabla 1. Por cada loncha y para cada tetraedro se envían los dos triángulos $\triangle P_0P_1P_2$ y $\triangle P_3P_2P_1$. Cada uno de esos seis vértices provocan una ejecución del programa de vértices, que desplaza el vértice a su posición correcta. Para ello, junto a cada vértice se ha de enviar las coordenadas de los vértices del tetraedro A, B, C, D, así como un índice del punto (0-3) y un identificador del triángulo al que pertenece el punto (0 para $\triangle P_0P_1P_2$ y 1 para $\triangle P_3P_2P_1$). Por último, también se envía la posición y_s de la loncha dentro del espacio de voxels que se quiere calcular. Esta posición es la que se utiliza para interpolar las posiciones de todos los vértices en programa de vértices.

La desventaja de esta solución es evidente. Para cada loncha es necesario enviar a la GPU la lista completa de tetraedros, codificada en un array de vértices. Este array almacena seis veces cada tetraedro (uno por vértice de $\triangle P_0P_1P_2$ y $\triangle P_3P_2P_1$), con el coste de memoria y de transferencia CPU-GPU que eso conlleva. Este array debe de mantenerse almacenado en memoria principal durante toda la ejecución del algoritmo.

Este esquema se muestra en la figura 2a.

5.2. Implementación en GPU mediante instanciación de geometría

Los problemas comentados en la sección anterior pueden superarse haciendo uso de las nuevas características de la serie 8 de NVIDIA GeForce. La técnica propuesta consigue alcanzar dos fines. Primero, minimizar el tamaño de las estructuras en memoria necesarias, reduciendo así el tráfico CPU-GPU. Y segundo, aprovechar las nuevas características del hardware para simplificar el algoritmo.

Para ello hacemos uso de la instanciación de geometría. En esta implementación, tan solo se envían al cauce gráfico dos triángulos por loncha. La GPU se encarga de generar tantas copias de los triángulos como sean necesarias (dos triángulos por tetraedro). En las coordenadas de cada vértice se indican un índice del punto (0-3) y un identificador del triángulo al que pertenece el punto (0-1). Este esquema se muestra en la figura 2b.

Nótese que todos los triángulos instanciados son iguales, por lo que no es posible pasar los tetraedros como parámetro de los vértices. En lugar de ello, se genera una textura 2D de flotantes que contiene todos los vértices de los tetraedros.

Esta textura almacena consecutivamente los vértices A, B, C, D de cada uno de los tetraedros del objeto a voxelizar, apareciendo ordenados en cada tetraedro por su coordenada y. Cada texel de la textura contiene tres flotantes (RGB), lo que permite almacenar las coordenadas X,Y,Z del vértice de un tetraedro. Para cada tetraedro se necesitan cuatro texels. El tamaño máximo de textura permitido nos limita la complejidad de la geometría que podemos representar

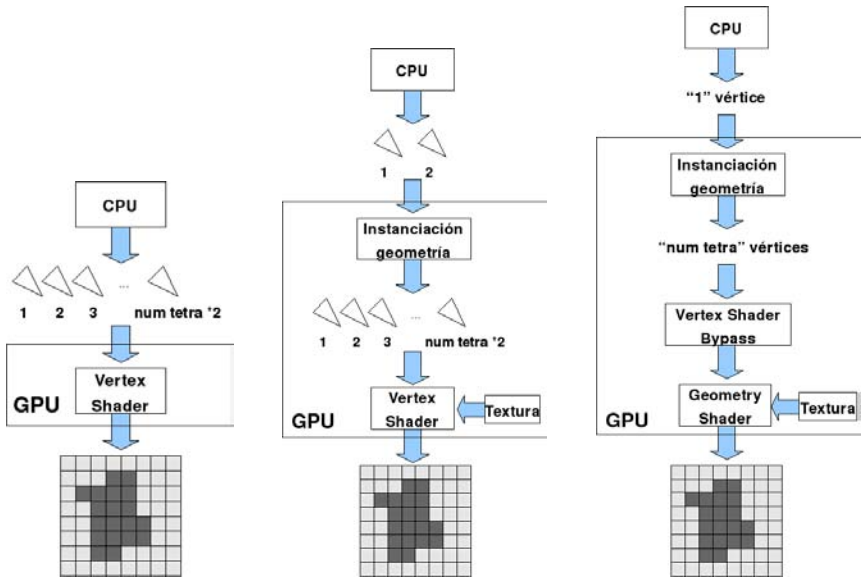

Figura 2: *Distintas soluciones estudiadas.*

en textura. Una textura de 4096^2 nos permite trabajar con modelos de más de cuatro millones de triángulos.

La textura es enviada una única vez a la GPU, en donde se almacena y es reutiliza para el procesado de cada loncha. De esta forma, tan solo se ha de enviar la geometría a la GPU una vez, y no sucesivas veces por cada loncha. Una vez enviada la textura a la GPU puede liberarse su espacio de la memoria principal.

Un programa de vértices recibe los puntos instanciados, así como el identificador de instancia *gl_InstanceID* para cada uno. Empleando este identificador como índice dentro de la textura, se extraen los vértices A, B, C, D correspondientes y los usamos para desplazar el punto a su posición correcta.

Esta versión del algoritmo funciona como sigue:

1. Crear la textura, que almacena los vértices A, B, C, D de cada tetraedro, y enviarla a la GPU.
2. Inicializar y_s al tamaño del espacio de vóxeles.
3. Crear un array de vértices que almacene únicamente dos triángulos, $\triangle P_0P_1P_2$ y $\triangle P_3P_2P_1$.
4. Limpiar el *framebuffer* y establecer la operación lógica por píxel a GL_XOR.
5. Establecer como variables uniformes y_s y el tamaño de la textura. Enviar el array de vértices a la GPU con *DrawArraysInstancedEXT*, indicando que se cree una instancia por cada tetraedro.
6. Copiar el resultado del *framebuffer* a memoria principal de la CPU.
7. Decrementar y_s y volver al paso 4 hasta $y_s = 0$.

5.3. Implementación en GPU mediante el procesador de geometría

Cuando un tetraedro no intersecta a la loncha actual, los triángulos $\triangle P_0P_1P_2$ y $\triangle P_3P_2P_1$ deben descartarse. Las implementaciones anteriores, debido a la imposibilidad de detener el cauce, recurrían a trasladarlos a posiciones alejadas para que fueran recortados en etapas posteriores.

Mediante el uso conjunto de la instanciación y del procesador de geometría, podemos solventar este problema. Este esquema se muestra en la figura 2c. El código completo en GLSL de esta propuesta se muestra en los listados 1 y 2.

En esta implementación, en cada loncha tan solo es necesario enviar a la GPU el tamaño de la textura, el valor y_s y un único vértice. Las coordenadas del vértice son intrascendentes. Dicho vértice se envía con la función *glDrawArraysInstancedEXT*, indicando que se instancie una vez por cada tetraedro (ver listado 1). Esto provoca una ejecución del procesador de geometría por cada tetraedro. Nótese que tan solo se puede acceder a la variable *gl_InstanceID* desde el procesador de vértices, por lo que necesitamos ejecutar un pequeño programa de vértices que copie ese valor a una de las coordenadas del vértice.

El programa de geometría del listado 2 toma como entrada un vértice, y genera como salidas:

- Dos triángulos, si $B_y > y_s \geq C_y$.

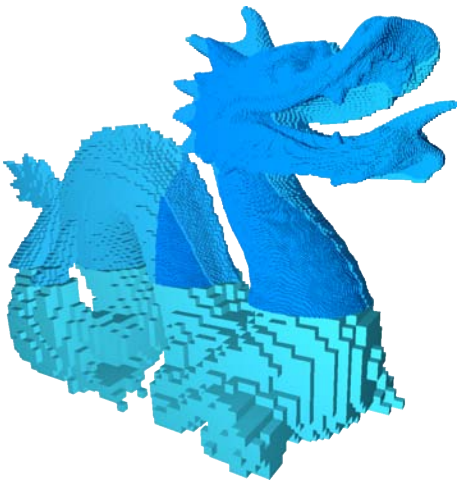

Figura 3: *Ejemplo de descomposición espacial (árbol-kd). Cada región se ha voxelizado a distinta resolución.*

- Un triángulo si $A_y > y_s \geq B_y$ o $C_y > y_s \geq D_y$
- Nada, si y_s no está entre A_y y D_y.

Los triángulos salen del procesador de geometría con sus vértices interpolados según la arista y loncha a la que pertenecen. La información de los tetraedros se recupera de textura, de forma análoga a la implementación de la sección anterior.

Si la loncha está comprendida en el intervalo $A_y > y_s \geq D_y$, se interpolan los valores de los vértices P_0, P_1, P_2 y se emiten, generando el triángulo $\triangle P_0P_1P_2$. A partir de los tres primeros vértices emitidos, cada nuevo vértice que se emita genera un nuevo triángulo. Por tanto, si la loncha está comprendida en el intervalo $B_y > y_s \geq C_y$, se interpola el cuarto vértice P_3 y se emite, dando lugar al triángulo $\triangle P_3P_2P_1$.

Gracias al procesador de geometría, los cálculos de interpolación que antes se hacían por vértice, ahora pueden optimizarse a nivel de primitiva. Esto permite que si el tetraedro no corta la loncha actual, puedan descartarse completamente los dos triángulos sin necesidad de enviar sus seis vértices fuera del volumen de vista. Tan solo se genera aquella geometría que es estrictamente necesaria, evitando cáculos innecesarios en etapas posteriores.

5.4. Mejoras

El uso de la instanciación de geometría nos permite realizar múltiples voxelizaciones de un mismo modelo sin necesidad de transferir su geometría a la tarjeta de vídeo más de una vez. Además, el algoritmo permite de forma sencilla voxelizar partes concretas del modelo a distinta resolución sin más que ajustar los parámetros de la cámara adecuadamente.

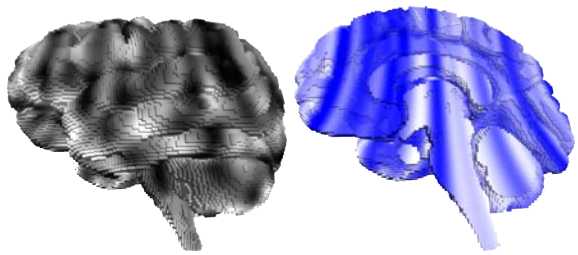

Figura 4: *Uso de un generador de ruido de Perlin para asignar un color a cada vóxel.*

Para restringir la voxelización a una región concreta del modelo es necesario:

- Fijar los planos que delimitan el volumen de visión para que las regiones que no se desean voxelizar queden fuera y sean recortadas por el hardware.
- Limitar el recorrido realizado por el plano de loncheado y_s al intervalo deseado.
- Variando el tamaño de la ventana de visión podemos ajustar la resolución de la voxelización.

Este método nos posibilita la construcción de estructuras de descomposición espacial (por ejemplo, los *octrees*) empleando la GPU y reduciendo el flujo de datos CPU-GPU a un solo vértice por loncha.

La figura 3 muestra un ejemplo de construcción de un árbol-*kd* en la GPU mediante instanciación de geometría. La resolución de la voxelización para cada nivel del árbol mostrado es 64, 128, 256 y 512.

Ambos métodos de voxelización descritos en este artículo permiten el uso de un programa de fragmentos para asignar propiedades a cada vóxel de forma eficiente. La figura 4 muestra el resultado de añadir a nuestro algoritmo un generador de ruido de Perlin 3D implementado en el procesador de fragmentos [Gre05].

6. Resultados experimentales

Hemos comparado las tres versiones del algoritmo (procesador de vértices, procesador de vértices con instanciación y procesador geometría). Para una comparativa de este algoritmo con otros algoritmos de distintos autores, véase [ORSF07]. Todas las implementaciones han sido realizadas en C++ y OpenGL Shading Language (GLSL), empleando el mismo compilador y optimizaciones. Las pruebas se ejecutaron en un Intel Core2 CPU 6600 2.40GHz con una GeForce 8800GTS bajo GNU/Linux. La versión del driver de NVIDIA empleada es la 169.09.

		64^3			128^3		
Modelo	**Triang.**	**Previo**	**Instan**	**Geom**	**Previo**	**Instan**	**Geom**
Torusknot	1200	0.0059	0.0082	0.0110	0.0173	0.0228	0.0273
Depot	10591	0.0241	0.0303	0.0500	0.0519	0.0654	0.1031
Bunny	69451	0.1340	0.1647	0.3522	0.2734	0.3241	0.7009
Dragon	871414	1.6386	1.8958	3.1970	3.2827	3.7640	6.3717
Buddha	1087716	7.5814	2.4506	3.9627	15.1674	4.7794	7.7655
Camera	2171026	15.3782	4.7577	7.8783	30.5920	9.3936	15.6449

		256^3			512^3		
Modelo	**Triang.**	**Previo**	**Instan**	**Geom**	**Previo**	**Instan**	**Geom**
Torusknot	1200	0.0773	0.0856	0.0970	0.5945	0.6043	0.6280
Depot	10591	0.1470	0.1724	0.2474	0.7371	0.7802	0.9216
Bunny	69451	0.5890	0.6832	1.4359	1.6176	1.7755	3.2680
Dragon	871414	6.6063	7.5347	12.7461	13.6529	15.7955	25.9143
Buddha	1087716	30.3766	9.4721	15.4752	61.2425	19.2438	31.2062
Camera	2171026	61.3413	18.6991	31.2532	122.9660	38.0506	63.0984

Tabla 2: *Tiempo de voxelización (en segundos) de las tres implementaciones GPU del algoritmo: en procesador de vértices (previo), con instanciación de geometría (Instan), y en procesador de geometría con instanciación (Geom). Las resoluciones de voxelización son* 64^3, 128^3, 256^3 *y* 512^3.

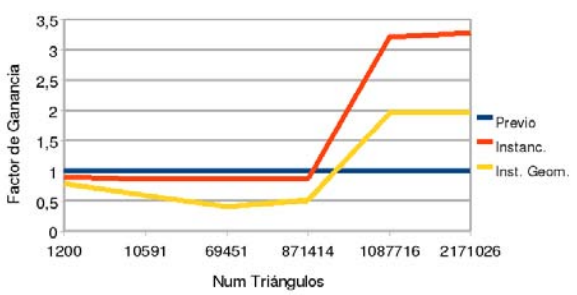

Figura 5: *Ganancias en tiempo para la resolución de* 256^2 *y los distintos modelos empleados.*

La tabla 2 muestra los tiempos invertidos por cada implementación en calcular la voxelización de los modelos de prueba a distintas resoluciones. La figura 6 muestra los modelos empleados.

La principal ventaja del uso de la instanciación de geometría es que se consigue reducir el tiempo necesario en subir el modelo en cada loncha, lo que repercute en mejores tiempos. La ganancia es mayor cuanto más complejo es el modelo a voxelizar.

La gráfica 5 muestra la ganancia obtenida por las dos implementaciones propuestas frente a la versión previa sin instanciación. Puede apreciarse como la versión previa se satura a partir del millón de triángulos. Ello se debe a que la necesidad de enviar toda la geometría en cada loncha constituye un cuello de botella al trabajar con modelos complejos. En cambio, con el uso de la instanciación de geometría se elimina esta necesidad, y el algoritmo ofrece mejores resultados en modelos complejos.

Nótese que la versión que emplea procesador de geometría no obtiene mejores tiempos pese a ser capaz de descartar del cauce gráfico aquellos tetraedros que no interesecten el plano de corte.

Se han realizado experimentos eliminado el acceso a textura de ambas soluciones propuestas y asignando valores fijos a las coordenadas de los tetraedros. En este escenario, los tiempos obtenidos por ambas soluciones son prácticamente idénticos. De ello se desprende que el acceso a textura desde el procesador de geometría está peor optimizado que desde el procesador de vértices. Estos experimentos también demuestran que descartar una primitiva del cauce gráfico no apor ta un beneficio considerable frente a dejarla terminar.

Otro detalle a considerar es el consumo de memoria. Ambas versiones que emplean instanciación almacenan la geometría del objeto en textura. Una vez enviada la textura a la GPU, el espacio de ésta puede liberarse de la memoria principal. Cada triángulo del objeto forma un tetraedro, y para cada tetraedro se debe almacenar sus cuatro puntos. Por ejemplo. el modelo del dragón (871414 triángulos) ocupa casi 40 MB (asumiendo que un flotante ocupa 4 bytes). Podemos almacenarlo en una textura RGB de flotantes de 2048^2, cuyo tamaño es de 48 MB. Puesto que la GeForce 8800 soporta texturas no cuadradas y de

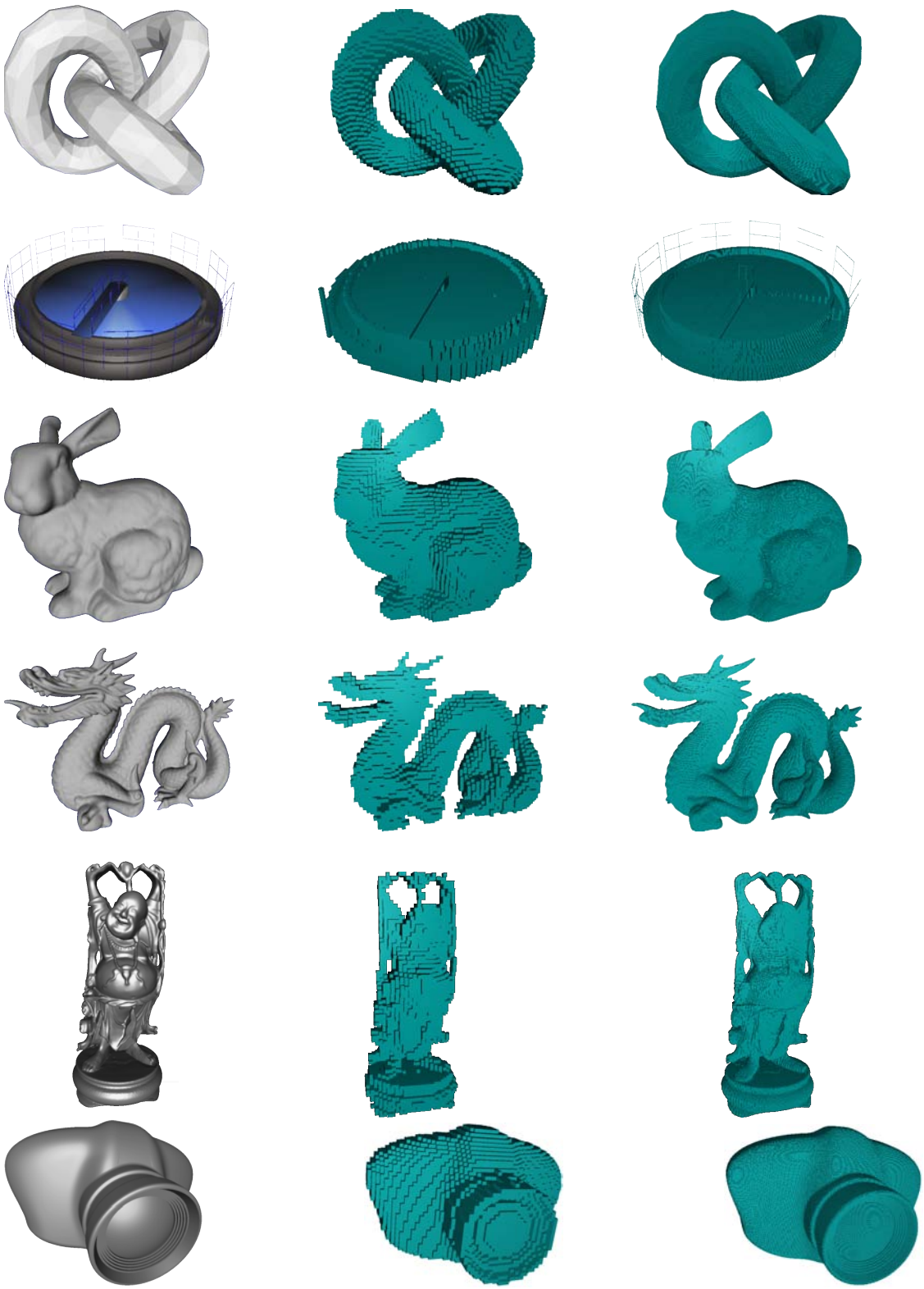

Figura 6: *Modelos usados en los experimentos. Se muestra la voxelización a resoluciones de 64 y 512.*

tamaño no potencia de 2, podemos ajustar el tamaño de ésta para minimizar el espacio desaprovechado.

En cambio, en la versión previa en GPU, tal y como se explica en la sección 5.1, se requiere almacenar un array de vértices que contenga la lista completa de tetraedros seis veces. Para el caso del dragón, son necesarios 239 MB, que no podrán liberarse mientras dure el proceso.

7. Conclusión

En este artículo hemos demostrado cómo el uso de las características aún poco explotadas de las nuevas tarjetas gráficas (instanciación de geometría, procesador de geometría) pueden ser empleadas para simplificar y potenciar algoritmos gráficos.

Gracias al procesador de geometría, podemos aplicar optimizaciones a nivel de primitiva. También se elimina la necesidad de emplear complejas y poco intuitivas estructuras para enviar información a la GPU (una de las principales dificultades que encuentran los programadores de GPUs).

El uso de la instanciación de geometría permite reducir en gran medida el flujo de datos CPU-GPU (uno de los cuellos de botella más importantes para muchas aplicaciones), lo que repercute positivamente en la eficiencia de aplicaciones como la voxelización. Además se ha reducido sustancialmente el consumo de memoria.

Es de esperar que estas mismas técnicas puedan emplearse en otro tipo de problemas geométricos obteniendo los mismos beneficios.

Agradecimientos

Este trabajo ha sido parcialmente financiado por la Junta de Andalucía y la Unión Europea (FEDER), a través de los proyectos P06-TIC-01403 y P07-TIC-02773; así como por el Ministerio de Educación y Ciencia de España y la Unión Europea (FEDER) a través del proyecto TIN2007-67474-C03.

Referencias

[BL06] Brown P., Lichtenbelt B.: *GL_EXT_geometry_shader4 extension documentation.* NVIDIA Corporation, 2006.

[Car05] Carucci F.: *Inside Geometry Instancing.* GPU Gems 2: Programming Techniques for High-Performance Graphics and General Purpose. Addison-Wesley, 2005, pp. 47–68.

[ED06] Eisemann E., Décoret X.: Fast scene voxelization and applications. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2006), ACM SIGGRAPH, pp. 71–78.

[FC00] Fang S., Chen H.: Hardware accelerated voxelization. *Computers and Graphics 24*, 3 (2000), 433–442.

[FT97] Feito F. R., Torres J. C.: Inclusion test for general polyhedra. *Computers & Graphics 21*, 1 (1997), 23–30.

[Gol06] Gold M.: *GL_EXT_draw_instanced extension documentation.* NVIDIA Corporation, 2006.

[Gre05] Green S.: *Implementing Improved Perlin Noise.* GPU Gems 2: Programming Techniques for High-Performance Graphics and General Purpose. Addison-Wesley, 2005, pp. 409–416.

[KCY93] Kaufman A. E., Cohen D., Yagel R.: Volume graphics. *IEEE Computer 26*, 7 (1993), 51–64.

[KPT99] Karabassi E.-A., Papaioannou G., Theoharis T.: A fast depth-buffer-based voxelization algorithm. *J. Graph. Tools 4*, 4 (1999), 5–10.

[LFWK05] Li W., Fan Z., Wei X., Kaufman A.: *Flow simulation with complex boundaries.* GPU Gems 2: Programming Techniques for High-Performance Graphics and General Purpose. Addison-Wesley, 2005, pp. 747–763.

[ORSF07] Ogáyar C. J., Rueda A. J., Segura R. J., Feito F. R.: Fast and simple hardware accelerated voxelizations using simplicial coverings. *Vis. Comput. 23*, 8 (2007), 535–543.

[PKT04] Passalis G., Kakadiaris I. A., Theoharis T.: Efficient hardware voxelization. In *CGI '04: Proceedings of the Computer Graphics International* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 374–377.

Listing 1: *Código C para configurar la ejecución del programa de geometría*

```
float s = y_max;

for(int ns=voxelSpace; ns>0; ns--, s-=intervalo){

  //establecer loncha y_S
  glUniform1f( slice, s );
  //establecer tamaño textura
  glUniform1f( textureSize, texSize );

  //enviar un punto por loncha
  glDrawArraysInstancedEXT(GL_POINTS, 0, 1, ntetra);

  //Transferir framebuffer a memoria

  //limpiar framebuffer
  glClear(GL_COLOR_BUFFER_BIT);

}
```

Listing 2: *Código GLSL del programa del procesador de geometría para la voxelización*

```
//GLSL version 1.20
#version 120

//New G80 extensions
#extension GL_EXT_geometry_shader4 : enable
#extension GL_EXT_gpu_shader4 : enable
#extension GL_ARB_texture_rectangle : enable

#define IN_INTERVAL(v, a, b) (a > v && v >= b)
#define INTERP(A,B,s)
  (mix (A, B, (s - A[1]) / (B[1] - A[1])))

uniform float slice;
uniform float textureSize;

uniform sampler2DRect text;

void main()
{
  vec4 tVertexA;    // Vertex 1 coordinates (bottom)
  vec4 tVertexB;    // Vertex 2 coordinates
  vec4 tVertexC;    // Vertex 3 coordinates
  vec4 tVertexD;    // Vertex 4 coordinates (top)

  float aux = gl_PositionIn[0].x * 4.0;
  float pos = floor(aux / textureSize);
  float offset = mod(aux, textureSize);

  tVertexA=texture2DRect(text,vec2(offset, pos));
  tVertexD=texture2DRect(text,vec2(offset+3.0,pos));

  gl_FrontColor = gl_FrontColorIn[0];

  vec4 p = vec4(0, 0, 0, 1);

  if(IN_INTERVAL(slice,tVertexA[1],tVertexD[1])){
    tVertexB=texture2DRect(text,vec2(offset+1.0,pos));
    tVertexC=texture2DRect(text,vec2(offset+2.0,pos));

    p.xy = INTERP(tVertexA, tVertexD, slice).xz;
    gl_Position = gl_ProjectionMatrix*p;

    //enviar P0
    EmitVertex();

    p.xy = (slice >= tVertexB[1]) ?
      INTERP(tVertexA, tVertexB, slice).xz :
      INTERP(tVertexB, tVertexD, slice).xz;
    gl_Position = gl_ProjectionMatrix*p;

    //enviar P1
    EmitVertex();

    p.xy = (slice >= tVertexC[1]) ?
      INTERP(tVertexA, tVertexC, slice).xz :
      INTERP(tVertexC, tVertexD, slice).xz;
    gl_Position = gl_ProjectionMatrix*p;

    //enviar P2
    EmitVertex();

    if(IN_INTERVAL(slice,tVertexB[1],tVertexC[1])){
      p.xy = INTERP(tVertexB, tVertexC, slice).xz;
      gl_Position = gl_ProjectionMatrix*p;

      //enviar P3
      EmitVertex();
    }
    EndPrimitive();
  }
}
```